Vladymir 2.1 – User Guide

Jonas Lätt

April 17, 2008

Contents

1 Introduction **3**
1.1 Vladymir in two words . . . 3
1.1.1 What is Vladymir ? . . . 3
1.1.2 What is Vladymir not ? . . . 3
1.1.3 Authors . . . 3
1.1.4 General License Agreement and Lack of Warranty . . . 3
1.2 Installation Guide . . . 4
1.2.1 Tested platforms . . . 4
1.2.2 Preparation: edit the Makefile . . . 4
1.2.3 Preparation: environment variables . . . 4
1.2.4 Unpacking and compiling . . . 4
1.2.5 Installation and de-installation . . . 4
1.2.6 Compiling and executing sample code . . . 5
1.3 Basics . . . 5
1.3.1 Array based programming . . . 5
1.3.2 Parallelization . . . 5

2 Sample application **5**

3 Constructors **8**
3.1 The Matrix class template . . . 8
3.2 Constructor for up to four dimensions . . . 8
3.3 General constructor . . . 9
3.4 Copy constructor . . . 9
3.5 Specifying a data manager at construction . . . 10

4 Initialization of data **10**
4.1 Initialization with constant value . . . 10
4.2 Initialization from raw data . . . 10
4.3 Initialization from a function object . . . 11
4.4 Further possibilities . . . 11

5 Matrix expressions **11**
5.1 Overloaded operators . . . 11
5.2 Reductions . . . 12
5.3 Conversions . . . 12
5.4 Meshgrids . . . 12

6 Indexing **13**
6.1 Memory layout . . . 13
6.2 Ranges and submatrices . . . 13
6.3 Matrix shifts . . . 14
6.4 Concatenations . . . 15
6.5 Permutations . . . 15

6.6 Dimension map . . . 15
6.7 Matrix view . . . 16

7 Domain specifications **16**
7.1 The `where` command . . . 16
7.2 Bool masks . . . 17
7.3 Local `where` conditions . . . 17

8 Input / Output **18**
8.1 Vladymir I/O streams . . . 18
8.2 Serialization and unserialization of a Matrix . . . 18
8.3 Binary file I/O . . . 19
8.4 Defining a new streamable device . . . 19
8.5 Using Matlab(tm) . . . 19

9 Copy semantics **20**

10 Parallelism **20**
10.1 Concepts . . . 20
10.2 Data managers . . . 20
10.3 Caveats . . . 21
10.3.1 Erroneous code . . . 21
10.3.2 Unefficient code . . . 21
10.4 Load balancing . . . 22
10.5 Dynamic load balancing . . . 22
10.5.1 Data organisation and load balancing . . . 22
10.5.2 The 'Group' class . . . 22
10.5.3 Selecting Algorithms . . . 23
10.5.4 Note on memory usage . . . 23
10.5.5 An example application . . . 24

11 Extending Vladymir **25**
11.1 Defining new operators . . . 25
11.2 Accessing raw data . . . 25

12 Optimizations **25**
12.1 Dead data . . . 25
12.2 Load sections . . . 26
12.3 Data alignment . . . 26
12.4 Reference or copy . . . 26

A Arithmetic and logical functions **27**
A.1 Overloaded operators . . . 27
A.1.1 Arithmetic binary functions . . . 27
A.1.2 Logical binary functions . . . 27
A.1.3 Bit-level binary functions . . . 27
A.1.4 Unary functions . . . 27
A.2 Math functions, mostly from cmath . . . 28
A.2.1 Unary functions . . . 28
A.2.2 Binary functions . . . 28
A.3 Reduction functions . . . 28

B Frequently asked questions **28**

1 Introduction

1.1 Vladymir in two words

1.1.1 What is Vladymir ?

Vladymir (**V**isual **la**ttice **dy**namics **m**atrix **i**nte**r**face) is a library for the C++ programming language that defines a new, multi-functional array type. The library is specifically designed for the purpose of array-based programming – a programming style in which the use of loops over array indices is replaced by simple expressions involving the array globally. Based on this seemingly restrictive programming paradigm, it offers the user two powerful features: straightforward programming style and automatic parallelization of the code on both shared and distributed memory environments. Although it certainly proves useful in many other contexts, the library has mainly been used and tested on the implementation of scientific models akin to Cellular Automata, thus implementing roughly nearest-neighbor dynamics. Here are some additional features that will make the heart of any scientific programmer beat faster:

- Symbiosis with Matlab(tm). During execution of the C++ program, data can be transferred to and from Matlab, exploiting for example the graphical aptitudes of this scientific tool.
- Complex boundaries. Although the interface of the library consists of objects behaving like traditional arrays, the internal implementation accounts for any kind of domain boundaries – optimizing both memory need and execution time.
- Dynamic load balancing. Especially in presence of complex boundaries, the work load of processors on a parallel implementation can be automatically outsmoothed.

Last but not least – Vladymir belongs to the open source community and may be used at no charge whatsoever.

1.1.2 What is Vladymir not ?

Vladymir is designed and optimized for a family of specific applications and is not a general-purpose library. In particular, it is not

- A library for doing linear algebra in C++. For urgent needs, you can use (non-parallel) Matlab(tm) functions, as they are made available by Vladymir. If that's not enough, forget Vladymir: you need a library that interfaces BLAS and LAPACK.
- An efficient implementation of arrays for brute-force computations. If this you want, you want Blitz++. This library uses so-called expression templates to achieve performances close to those of Fortran.
- A general-purpose parallelized numerical library. For this, check out POOMA.

1.1.3 Authors

- **Jonas Lätt** *Jonas.Latt@cui.unige.ch* . Main author. Corresponding author for criticism, suggestions and bug reports (please send me nice applications of Vladymir if you have some!).
- **Fokko Beekhof** Implementation of the dynamic load balancing strategies. Suggestions on the global class design.

1.1.4 General License Agreement and Lack of Warranty

All Vladymir source code files are Copyright (C) 2004 by Jonas Lätt (for contact, please use electronic mailing: *Jonas.Latt@tufts.edu*).

Vladymir is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

Vladymir is distributed in the hope that it will be useful, but *without any warranty*; without even the implied warranty of *merchantability* or *fitness for a particular purpose*. See the GNU Lesser General Public License for more details.

A copy of the GNU Lesser General Public License should be delivered along with Vladymir; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA.

1.2 Installation Guide

1.2.1 Tested platforms

The library has been successfully compiled and executed on the following platforms:

- Operating Systems. Ubuntu 8.04; Linux kernel 2.6.24
- Compilers. GNU gcc version 4.2.3
- MPI implementations. MPICH version 1.2.7 ; OpenMPI will *not* work!
- Matlab(tm) version. Version 7.0.0.19901 (R14)

Note that the current version is expected to compile only on operating systems of the Unix family. This is because of the platform specific implementation of file I/O. If need should arise, this part is however easily extended to other platforms.

Due to the use of advanced features of templates, the code will fail to compile on some older compilers.

1.2.2 Preparation: edit the Makefile

Some variables need to be set in the Makefile. Certain options can be set to your personal preference, such as which compiler to use or where to install Vladymir; other reflect your system's configuration, such as where to find Matlab headers.

1.2.3 Preparation: environment variables

A series of environment variables must be set or extended in order for compilation of Vladymir. Subsequently, when you use the Vladymir library, these environment variables must also be set, unless you decide to change the sample Makefiles correspondingly. Therefore, you might as well define those variables in your shell configuration script (.cshrc, .bashrc, ...). Here follows a list of needed environment variables:

`PATH`	add an entry to the Matlab(tm) binaries (if it's not already there); Example: `$MATLAB_DIR/bin`
`LD_LIBRARY_PATH`	add an entry to the Matlab(tm) runtime libraries; Example: `$MATLAB_DIR/bin/glnx86`

1.2.4 Unpacking and compiling

1. Download and unpack the tarball.
   Example: `tar jxvf vladymir-15.11.04 vladymir-15.11.04.tar.bz2`. Change into the Vladymir directory (which, by now, must be identical with `VLADY_DIR`.
2. If you lack MPI or Matlab(tm), you must compile Vladymir without support for those. Edit the Makefile and uncomment the corresponing lines (note that there exist free implementations of MPI, for example MPICH).
3. Compile Vladymir by typing `make`.

1.2.5 Installation and de-installation

1. After compiling Vladymir, type `make install` to install, or potentially `sudo make install` if you need superuser privileges.
2. To uninstall, type `make uninstall` or `sudo make install` if you need superuser privileges.

1.2.6 Compiling and executing sample code

There are some sample programs in the `sample` subdirectory, with a corresponding Makefile. For example, type `make wave` for compilation of the wave propagation example.

The programs can be executed in serial (type `wave`) or in parallel (type `mpirun -np 3 wave`).

1.3 Basics

1.3.1 Array based programming

The array based programming paradigm is useful for the simulation of spatially extended systems. The data is stored in a one- or multidimensional array. Examples are the grid nodes in a numerical simulation of a physical system, the pixels of an numeric image, or the the individuals of a population in a genetic algorithm. The essense of this programming style is that all elements of the array are treated in the same manner throughout the program execution. In other terms, the operations applied on the data are executed in a single instruction flow.

Let us illustrate this idea by a simple example. We have two arrays `a` and `b` and want to add to each element of `a` the square root of the corresponding element of `b`. With Vladymir, this operation looks as if `a` and `b` were scalar variables:

```
a += b;
```

Here, the loop over the array indices is implicit. In an analoguous way, more complicated expressions are formulated, for example to change the size of an array or the relative arrangement of array data.

Although the paradigm seems fairly restrictive, it proves useful in a large area of applications. The widely used scientific programming tool Matlab(tm) makes extensive use of it, and there has even been produced dedicated hardware for an execution of array-based code that is parallel at instruction level.

1.3.2 Parallelization

The array-based formulation of a program is efficiently exploited to implement parallelism at data level (as opposed to instruction level). To do this, all data contained in Vladymir data structures is distributed over the nodes of a parallel machine, and the single instruction flow is executed on each one of those nodes.

Technically, a C++ program that uses the Vladymir library is executed on every node of a parallel machine, and communication between those instances of the program is achieved through the Message Passing Interface (MPI) standard. At run time, C++ code that is based on non-Vladymir datatypes (*e.g.* native datatypes) is duplicated on all nodes and thus executed at the speed of one single node. Parallel speed up is obtained only at the execution of array-based code.

A surprising fact for the Vladymir user is that there is no need for writing explicitly parallel code nor even for knowing anything about parallelism. Parallelization is automatic and is implemented at run-time, without the need for any specialized compiler or pre-compiler. The only requirement made on the user is that she or he understands and respects the philosophy of array-based programming in order to aid Vladymir in parallelization tasks.

There exist however possibilities for the user to steer Vlaymirs parallelization strategies. This happens through policy classes that define the layout of the data over the parallel machine. Vladymir offers a set of default policies that implement both static and dynamic load balancing strategies.

2 Sample application

In this section, we will go through a complete Vladymir application step by step. The application should be relatively easy to follow and gives a brief overview of the philosophy and the keynote features of Vladymir. At the same time, it makes clear that Vladymir is extremely expressive and straightforward for the formulation of scientific problems.

The program implements a numerical simulation of wave propagation. It describes a diffraction experiment through a wall with two slits. The model is based on a so-called lattice Boltzmann method. The wave is described not only by one scalar value (the amplitude), but by four quantities, representing a kind of particle streams along the four directions of a square lattice. The space is discretized on a regular

grid, and the grid notes are represented by a real-valued Vladymir Matrix. Thus, the fundamental variables of the program are four matrices *f0*, *f1*, *f2* and *f3*. The evolution of the system is governed by nearest-neighbour dynamics: at every time step, the values of a grid node are combined with the values of its four neighbour nodes only. In this sense, the lattice Boltzmann model is a real-value extension to Cellular Automata.

```
#include "vladymir/base/pvladymir.h"                                    /*(0)*/

const double      pi = 3.14159;
const int         l = 200, h = 200;                                     /*(1)*/
const int         wp = 90, h1 = 69, h2 = 110, hw = 20;                  /*(2)*/
const double      A = 1;                                                /*(3)*/
const double      omega = 0.03*2*pi;                                    /*(4)*/

int main(int argc, char *argv[]) {
    iniVladymir(&argc, &argv);                                          /*(5)*/

    fMatrix f0(l,h), f1(l,h), f2(l,h), f3(l,h);                         /*(6)*/
    fMatrix f_0(l,h), f_1(l,h), f_2(l,h), f_3(l,h);
    fMatrix psi(l,h);                                                   /*(7)*/
    bMatrix wall(l,h);                                                  /*(8)*/

    f0 = 0.;                                                            /*(9)*/
    f1 = 0.;
    f2 = 0.;
    f3 = 0.;

    wall = false;
    wall[range(wp,wp+1)] = true;                                        /*(10)*/
    wall[range(wp,wp+1)][range(h1,h1+hw-1] = false;                     /*(11)*/
    wall[range(wp,wp+1)][range(h1,h1+hw-1] = false;

    for (int t = 0; t <= 280; ++t) {                                    /*(12)*/
        f0[0]     = A*sin(omega*t);                                     /*(13)*/
        f2[l-1]   = 0.;

        psi.assign(f0 + f1 + f2 + f3);                                  /*(14)*/

        if (t % 10 == 0) {
            toMatlab(psi*(!wall.convert<double>()), "psi");             /*(15)*/
            matlab().command("imagesc(psi)");                           /*(16)*/
        }

        where(wall);                                                    /*(17)*/
            f_0.assign(-f2);                                            /*(18)*/
            f_1.assign(-f3);
            f_2.assign(-f0);
            f_3.assign(-f1);
        elsewhere();                                                    /*(19)*/
            f_0.assign(1./2.*psi-f2);                                   /*(20)*/
            f_1.assign(1./2.*psi-f3);
            f_2.assign(1./2.*psi-f0);
            f_3.assign(1./2.*psi-f1);
        endwhere();                                                     /*(21)*/

        f0.assign(f_0[rshift()]);                                       /*(22)*/
```

```
        f1.assign(f_1[all()][rshift()]);                    /*(23)*/
        f2.assign(f_2[lshift()]);
        f3.assign(f_3[all()][lshift()]);
    }
    toMatlab(sqr(psi[l-30]), "psi");                        /*(24)*/
    matlab().command("plot(psi)");
    cout << "enter a number to end program" << endl;
    int tmp; cin >> tmp
}
```

(0) At inclusion of the header file, you decide if the default definitions are adapted to a parallel (pvladymir.h) or serial (vladymir.h) program execution.

(1) The simulation is going to be executed on a grid of size $l \times h = 200 \times 200$.

(2) The wall will be located at the position $x = wp$, and there will be two holes of size `hw` at position $y = h1$ and $y = h2$.

(3) `A` is the amplitude and

(4) `omega` the frequency of the wave.

(5) This shall be the first line of code in the main() function whenever you utilize (serial or parallel) Vladymir. It is responsible or initialization of the parallel machine and I/O facilities.

(6) The `fMatrix` is a typedef that instantiates a `Vladymir::Matrix` template on double precision real numbers with an appropriate data manager. In a parallel execution, the default parallelization policy distributes the Matrix data with equal load over the nodes. The present version of the Matrix constructor takes two integer arguments (width, height) for constuction of a two-dimensional matrix. Other constructors are available for initialization of an arbitrary number of dimensions (section 3). We declare four matrices `fx` to represent the components of the LB wave, and four temporary matrices `f_x`.

(7) `psi`, the wave amplitude.

(8) `wall`, a boolean matrix that defines the double slit wall. The matrix will steer the behaviour of the wave: wave reflection where `wall`=1 and free propagation dynamics elsewhere.

(9) Initial condition of the `fx` matrices: all elements are set to 0. Note how a loop over space inidices is replaced by a collective operation.

(10) This is the first example of an array subindexing (in Fortran 90, this is called an array section). This expression assigns the value `false` to a column of width 2. A similar expression in, say, Matlab would have the following form: `wall`(`wp`+1:`wp`+2, :) = false.

(11) A hole in the wall is defined. This needs subindexing in both x-direction (first bracket) and y-direction (second bracket). The second bracket was left out in item **(10)**, meaning that all elements in y-directions are kept. This way of doing, skipping the joker `all()` in right-hand extreme brackets alieviates the notation and allows for more efficient execution.

(12) The time loop.

(13) Generation of a wave at the first column of the grid. Note that because boundary conditions are periodic (item **(22)**), the wave must be explicitly prevented from entering the field at the last column.

(14) The wave amplitude `psi` is the sum of the four wave components. In this expression, it is possible to replace the method `.assign` by the more intuitive assignment operator (`psi` = `f0` +...). This however would be slightly less efficient, as it involves construction of a temporary matrix, and it is generally a bad habit in this type of context (cf. item **(22)** and section 9).

(15) Graphical output using Matlab(tm). The data of a matrix is copied to the data space of Matlab(tm). Note the conversion operator that is applied on the matrix `wall`. In Vladymir, type conversions are not automatic, and types are requested to match exactly in an expression.

(16) Any string which represents a valid Matlab(tm) command may transmitted and executed at run time.

(17) Domain restriction. The *where* command is similar to the one used in Fortran 90. Arguments to the *where* command are a boolean Matrix or a so-called BoolMask (section 7). From this line on, all expressions evaluated upon matrices that have the same extent as the boolean condition matrix will be applied on the specified restricted domain only.

(18) Reflexion of the wave on the boundaries (on solid boundaries, the amplitude of a reflected wave is inverted).

(19) Inversion of the *where* condition (this is a constant time operation!)

(20) Free wave propagation in the very elegant LB formalism.

(21) End of domain restriction.

(22) Propagation step with nearest-neighbor dynamics. The matrix *f0* for example is shifted by one element in direction of increasing indices along x-dimension. In this example, the use of the method *.assign* instead of the assignment operator *operator=* is important. Indeed, the use of the assignment operator would be a bug, as it implies reference semantics on the assignment (cf. section 9). The result would be that both matrices *f0* and *f_0* reference the same data space. This is a behaviour which is typical for languages akin to Java; in the present context, it is not what we want.

(23) Similarly, the matrix *f1* is shifted by one element in direction of increasing indices along y-dimension.

(24) At the end, we plot the energy profile along a column. At program execution, you will be able to note peaks typical for diffraction experiments.

You can now compile the program (`make diffraction`) and run it on one (`diffraction`) or, say, three processors (`mpirun -np 3 diffraction`).

3 Constructors

3.1 The Matrix class template

Vladymir offers three default Matrix data types:

```
iMatrix  Matrix of integer values
fMatrix  Matrix of double precision real values
bMatrix  Matrix of boolean values
```

These types are really specializations of the generic template Matrix type. For example, *iMatrix* is defined as

```
typedef Matrix<int, DefaultDataManager> iMatrix.
```

The first template parameter defines the type of the Matrix data, and the second parameter the (parallel) data distribution policy. Even without any knowledge of C++ templates, it is easy to extend these definitions. For example, if you want to use a Matrix of char, you might write a typedef similar to the following:

```
typedef Matrix<char, DefaultDataManager> cMatrix.
```

For a basic use of Vladymir, this is about all you have to know about templates and data managers. More advanced knowledge is required especially when you want to optimize the load balance of a parallel run. Section 10.4 explains how this is done.

3.2 Constructor for up to four dimensions

Matrices can be constructed by default:

```
iMatrix sampleMatrix     // Default construction of an integer-valued matrix.
```

Through optional arguments to the constructor, memory can be allocated. The following command constructs a 2D matrix:

```
iMatrix sample2d(3,3)     // Construction of a two-dimensional 3 by 3 matrix
```

and the following command a 4D matrix:

```
iMatrix sample4d(2,3,4,5)     // Construction of a four-dimensional 2 by 3 by 4 by 5 matrix.
```

It is also possible to reshape a Matrix after construction:

```
iMatrix reshapedMatrix;
reshapedMatrix = iMatrix(3,3)   // The matrix is reshaped to be 2d 3 by 3.
```

3.3 General constructor

For construction of a Matrix with more than four dimensions, no specific overloaded constructor is available. In this situation, you must use the generic constructor that allows for construction of a Matrix with arbitrary shape. The argument to this constructor is a vector<int> which specifies the extent of the Matrix in every dimension. The number of elements in this vector equals the number of dimensions of the Matrix:

```
vector<size_t> extents;      // The extents of the Matrix to be constructed
extents.push_back(l0);       // l0, l1 and l2 are integer constants
extents.push_back(l1);
extents.push_back(l2);
fMatrix matrix3d(extents);  // Construction of a 3d Matrix
```

The 0-th element of the extent vector (extents[0] in our example) indicates the extent of the Matrix along the 0-th dimension, which is indexed by the right-most bracket. This means that the Matrix in the previous example could have been initialized in the following manner as well:

```
fMatrix matrix3d(l2,l1,l0);     // Construction of a 3d Matrix
```

and the elements of the matrix are indexed in the same order:

```
matrix3d[i2][i1][i0] = 0.5;
    // Assignment of a Matrix element with indices 0 <= i0 < l0 etc.
```

The 0-th dimension is defined as the dimension at which continuous increase of the index corresponds to a continuous increase in memory. This point is developed in section 6.1.

3.4 Copy constructor

A Matrix can be copy-construted from another Matrix of same type:

```
fMatrix a (3,3)
fMatrix b (a)    // b is copy constructed from a
```

You should however be aware that during copy construction, no data are really copied from `a` to `b`. Instead, due to *reference semantics* of copy construction in Vladymir, `a` and `b` are now two equivalent references to the same data. If you want to produce two independent copies of the Matrix, the data must be copied, for example like the following:

```
fMatrix a (3,3)
fMatrix b (a.clone())    // a and b hold two independent copies of the data
```

A final remark concerns the case when you want to construct a Matrix of same shape as an existing one, without copying data. This is done by requesting the extent vector from the latter Matrix:

```
fMatrix newMatrix(oldMatrix.getExtent());    // newMatrix has the same shape as oldMatrix.
```

In the same way, the methods Matrix<>::size() and Matrix<>::getDim() are used to request the total size and the number of dimensions of a Matrix.

3.5 Specifying a data manager at construction

Each Matrix possesses a fully constructed data manager. The data manager knows about the size and shape of the matrix and is responsible for access to the Matrix memory space. It offers an abstraction to the serial / parallel layout of the data and takes care of data regulation tasks such as domain restrictions (section 7) and so-called dead data (section 12.1).

For almost all common tasks, there is no need to worry about data managers, as their implementation is completely transparent. In some special cases one might however want to construct a Matrix from a specific data manager. Such a need appears for example when a group of Matrices need to share a common data manager, to synchronize their domain restriction or load balancing strategies (such as the strategy explained in section 10.5).

The following example defines a data manager for a two-dimensional data and constructs a Matrix from it. It uses the default data manager. A choice of available data managers is listed in section 10.2.

```
vector<size_t> extent (2);
extent[0] = 2;
extent[1] = 3;
// Data manager contains:
// - A dimension manager (with extent vector),
// - A data manager.  We take the global domain manager,
// which makes the Matrix subject to global where condition.
EquiBalancedDataManager<double> dMan(
    SPDimensionManager(
        new DimensionManager(extent)
    ),
    globalDomainManager()
);

fMatrix a(dMan);
```

4 Initialization of data

4.1 Initialization with constant value

Upon construction of a Matrix, the data are uninitialized (except for the case of copy construction). The simplest way to initialize them is through assignment of a constant value:

```
fMatrix a (3,3);
a = 3.;
```

This syntax is a shortcut for assignment of the value 3 to all 9 elements of the Matrix. As always in Vladymir, loops over space directions are replaced by a corresponding array-based operation.

4.2 Initialization from raw data

The Matrix values can also be initialized from raw data (*i.e.* pure C or C++ arrays). In that case, a pointer to the data is delivered to the Matrix:

```
fMatrix a(3,3);
double data[9] = { 1.,1.,1., 2.,2.,2., 1.,1.,1.  };
a.assignRaw(data);     // the data are fed to the Matrix, column by column.
```

The above code yields a correct result, in both a serial and a parallel execution. Note however that in the parallel case, only the data defined on the first processor are considered, while on the other processors, the pointer passed to the function assignRaw() is ignored. In general, this fact is innocent and can safely be ignored, but it is important to rember it when reading data from an external device. Of course, data I/O from streams (section 8.2) and from binary files (section 8.3) is handeled in way that is automatic and transparent to the user.

4.3 Initialization from a function object

The data of a Matrix can be initialized by a generic function using the forAll statement. The argument to this statement is a function or (better) a function object that assigns values to each element. The following example shows how to initialize the values of a fMatrix with random values between 0 and 1 using the standard function random():

```
// Definition of a function object for random values
struct IniRandom {
  void operator()(double& el) const {
    el = static_cast<double>(random()) / RAND_MAX;
  }
};
...
fMatrix a(3,3)
forAll(a, IniRandom());
```

4.4 Further possibilities

The strategy developed in the previous section does not enable to chose initialization values that are dependent on the matrix index. A convenient way of doing so is to use mesh grids. This is explained in section 5.4. The possibility of steering data initialization from files or streamable devices is described in section 8.

5 Matrix expressions

5.1 Overloaded operators

The Matrix type is defined to behave in a manner similar to integrated datatypes of the C++ language. Various operators have been overloaded in order to enable arithmetic and logical expressions of all kinds:

Program snippet:	Output:
`iMatrix a(3,3), b(3,3), c(3,3);`	`a =`
`b = 2;  c = 3;`	`8  8  8`
`a = b*(c+1);`	`8  8  8`
`cout << "a=" << endl;`	`8  8  8`
`a.print();`	
`a /= 2.;`	`a/2 =`
`cout << endl << "a/2=" << endl;`	`4  4  4`
`(a/2).print();`	`4  4  4`
	`4  4  4`

The same procedure applies to the evaluation of boolean expressions:

Program snippet:	Output:
`bMatrix a(3,3), b(3,3), c;`	`a||b =`
`a = true;  b = false;`	`1  1  1`
`c = a||b;`	`1  1  1`
`cout << "a||b=" << endl;`	`1  1  1`
`c.print();`	

Boolean expressions are useful for conditional evaluation of Matrix expressions. This multi-dimensional extension to the **if** condition is explained in section 7.

Arithmetic and logical expressions yield a temporary return type that is assignable to a Matrix. This return type is efficient in the sense that it does not copy the comlete matrix data from its arguments.

Therefore, evaluation of the expressions is fast and does not suffer much from the object oriented overhead. For maximum efficiency, the assignment operator (operator=) should be replaced by the method .assign(). Be aware however that, when doing so, you are responsible for allocating the right amount of memory for the Matrix to which data are assigned. More details on this topic are found in section 9.

A complete list of overloaded operators is given in Appendix A.1. Furthermore, a list of redefined functions from cmath (sqrt, cos, *etc.*) is given in Appendix A.2.

5.2 Reductions

Reduction functions offer a way to obtain a scalar value from a global reduction operation over the Matrix elements. Reduction functions are sum, product, locical and, *etc.* (cf. Appendix A.3). In the following example, a global sum of a Matrix is computed:

Program snippet:

```
iMatrix a(3,3); a = 2;
int sum = a.reduceSum();
cout << "The sum is " << sum << endl;
```

Output:

```
The sum is 18
```

Note that after reduction, the result is known on all nodes of the parallel machine. Reduction functions are actually the only means of obtaining a scalar value (*i.e.* a primitive C++ variable) from a Matrix. In particular, you need to use one (any one) of them to extract a single element from a Matrix:

```
fMatrix a(3,3);
double el = a[1][1];              // Error:  can't assign a Matrix to a scalar
double el = a[1][1].reduceSum();  // OK: Using a reduction operation
```

5.3 Conversions

There are no implicit conversions between Vladymir matrices. Even straightforward conversions such as from bool to int need to be stated explicitly. One reason for this implementation choice was that type conversions on big matrices can take a great amount of computation time. If they were implicit, the programmer could easily oversee them and incur (often avoidable) performance penalties. In the following example, a integer-valued Matrix is added to a real-valued one:

```
iMatrix iMat(3,3); iMat = 2;
fMatrix fMat(3,3); fMat = 2.;
fMat += iMat.convert<double>();  // Adding an fMatrix and an iMatrix
```

Internally, type conversions are peformed on base of the static type (*i.e.* through a static_cast()).

5.4 Meshgrids

In section 4.3, a method was introduced for initialization of the Matrix data from a generic function which is independent of the index of the initialized element. Such a dependency can be obtained through the use of mesh grids. A mesh grid is a iMatrix mg with elements defined in such a way that

$$mg_d[i_n][i_{n-1}]\cdots[i_d]\cdots[i_0] = i_d, \qquad \text{for } d \text{ a given dimension of the Matrix.}$$

In a two-dimensional case, this becomes:

Program snippet:

```
fMatrix a(3,3);
cout << "Meshgrid along x-direction:" << endl;
a.meshGrid(1).print();
cout << "Meshgrid along y-direction:" << endl;
a.meshGrid(0).print();
```

Output:

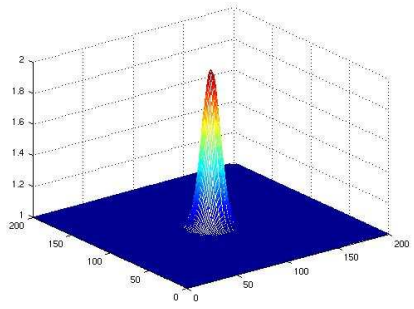
```
Meshgrid along x-axis:
0  1  2
0  1  2
0  1  2

Meshgrid along y-axis:
2  2  2
1  1  1
0  0  0
```

The following example shows how to use mesh grids for data initializations. It initializes a real-valued scalar field with a Gaussian peak in the center:

Program snippet:

```
const int l = 200;
const int h = 200;
const double A = 0.1;
fMatrix field(l,h);
fMatrix x = field.meshGrid(1).convert<double>()
fMatrix y = field.meshGrid(0).convert<double>()
// central Gaussian peak
field = 1.  + exp(-A*(sqr(x-l/2)+sqr(y-h/2)));
// Graphical output in Matlab (cf.  section 8.5)
toMatlab(field, "field");
matlab().command("surf(field)");
```

Output:



6 Indexing

6.1 Memory layout

The memory layout follows the C and C++ convention, which is usually referenced as *row-major order*, as opposed to the *column-major* order of Fortran arrays. However, this terminology is confusing in the present context, and we will not use it further (the confusion comes from the term *row-major* being based on the mathematical ij interpretation of the matrix indices, wheras Vladymir encourages the more physical xy interpretation. Those two interpretations disagree on what is a row and what is a column).

Whatever the terminology, Vladymir implements a memory layout such that a loop over the rightmost Matrix index sweeps over contiguous blocks in physical memory space most of the time. Note that this is a mere guideline, as Vladymir offers no strict guarantees on memory layout. Indeed, the internal data structure is rather a list of consequent memory chunks. As a consequence, the identity

$$matrix[x][y+d] = *(\&matrix[x][y]+d)$$

is wrong in the general case. This should seem obvious to the reader, at least for a parallel program execution where the data are distributed over several nodes.

The dimensions are numbered from right to left: the right-most bracket indices dimension 0, wheras the left-most bracket indices dimension n−1, where n is the total number of dimensions of the Matrix. This is the convention taken when initializing a Matrix with an extent vector: extent[0] is the extent of the Matrix in dimensions 0 *etc.*

Now for a completely independent topic: how are the matrix indices *interpreted* when a Matrix is printed to the console throught the methode .print()? The answer appears from the example shown in section 5.3. In two dimensions, the indices are interpreted as physical space indices, the left index (dimension 1) extending along the x-direction, and the right index (dimension 0) extending along the y-direction.

6.2 Ranges and submatrices

The simplest form of Matrix indexing uses integer indices. Each integer index reduces the dimensionality of the Matrix by one. When the number of indices used reaches the number of dimensions of the Matrix, a 0-dimensional Matrix (a kind of a scalar) is left:

Program snippet:

```
iMatrix m3(3,3,3);
m3 = m3.meshGrid(2);
m3[0][1][2].print();
```

Output:

```
0
```

However, this 0-dimensional Matrix is *not* convertible to a scalar:

```
int el = m3[0][1][2];  // Error:  cannot convert Matrix to int!
```

This is because the number of dimensions of a Matrix is not known at compilation time; it can therefore not be determined if it is convertible to a scalar. To obtain the conversion to a scalar, just apply (any) reduction operation (section 5.2):

```
int el = m3[0][1][2].reduceSum();  // Ok:  reduction returns a scalar.
```

In order to leave a dimension unchanged, one can apply the index all() that, in the same way as the joker (:) in Matlab(tm), addresses all elements in the corresponding direction. The following example reduces a Matrix over dimension 1:

Program snippet:

```
iMatrix m3[3][3][3], m2[3][3];
m3 = m3.meshGrid(2);
m2.assign(m3[all()][1][all()]);
m2.print();
```

Output:

```
2  2  2
1  1  1
0  0  0
```

The `all()` specification on the rightmost indices is optional. Thus, the third line in the previous example could as well have been written as follows:

```
m2.assign(m3[all()][1]);
```

In the general case, the scalar index and the `all()` specification are replaced by a range argument. This is done through a range() function that spans a contiguous range in a given direction:

Program snippet:

```
iMatrix a(3,3);
a = a.meshGrid(1);
a[range(0,1)].print();
cout << endl;
a[range(2,0)].print();
```

Output:

```
1  1  1
0  0  0

0  0  0
1  1  1
2  2  2
```

Those ranges can be combined in an arbitrary way to extract submatrices from a Matrix:

```
iMatrix a(7,8,9,10);
iMatrix b(3,3);
b.assign(a[0][range(5,7)][range(1,3)]);
```

6.3 Matrix shifts

Matrix shifts perform a shift of n elements in positive (rshift(**int** n)) or negative (lshift(**int** n)) direction. The boundaries are periodic, which means that elements that are shifted out of one boundary enter the opposite boundary. The default value for n is 1:

Program snippet:

```
iMatrix xMatr(4,3);
xMatr = xMatr.meshGrid(1);
xMatr[all()][rshift()].print();
cout << endl;
xMatr[all()][rshift(2)].print();
```

Output:

```
3  0  1  2
3  0  1  2
3  0  1  2

2  3  0  1
2  3  0  1
2  3  0  1
```

6.4 Concatenations

The indexing expressions discussed in the previous sections (and in the next section) can be combined to produce the most general form of a matrix permutation. This is obtained by aligning (using the function `cat`, in the case of two to four arguments) several index expressions. The resulting matrix can have any size, even larger than the size of the original matrix. Note that a scalar index must be explicitly instantiated using the expression `singleIndex(i)`. The resulting Matrix can have any size, even larger than the size of the original Matrix:

Program snippet:

```
iMatrix xMatr(2,2);
xMatr = xMatr.meshGrid();
xMatr[cat(all(), range(1,0),
singleIndex(0))].print();
cout << endl;
```

Output:

```
0  1  1  0  0
0  1  1  0  0
```

When aligning more than four indexing operators, a `CatIndex` must explicitly be constructed:

```
shared_ptr<CatIndex> ci(new CatIndex);  // Creating a new CatIndex
                                        // (must be encapsulated in a shared_ptr)
ci->add(all());                         // adding arbitrary number of
                                        // indexing expressions
ci->add(range(1,0));
ci->add(singleIndex(0));
...
matr[ci].print();                       // Appplying the concatenated indices
```

6.5 Permutations

This feature is yet to be implemented in Vladymir...

6.6 Dimension map

Suppose you want to apply the same type of sub-indexing to a group of matrices and don't want to type the same expression all over again. Or, as another problem, suppose you can't type your indexing statically, as the number of dimensions in your Matrix is known at run time only. In both cases, a solution is to store the expression in a `DimMapping` object. Here is how to proceed:

- Initialize the `DimMapping` object. The constructor needs to know the extent of the Matrix along every dimension.

  ```
  vector<size_t> extent;
  extent.push_back(3);    // dimension 0 (y)
  extent.push_back(3);    // dimension 1 (x)

  DimMapping dimMap(extent);
  ```

- In each dimension, indicate the index expression you want to apply. There is no default initialization, so it is mandatory to initialize *all* dimensions.

  ```
  dimMap.addIndex(0, *all());
  dimMap.addIndex(1, *rshift());
  ```

- Apply the mapping

  ```
  fMatrix a(extent);
  forAll(a, IniRandom());
  a[dimMap].print();
  ```

6.7 Matrix view

There is a fundamental difference between objects returned from algebraic/logical expressions as in the following

matr1 = *matr2*+*matr3*; // *matr1*,$\cdots$ are of type Matrix<>

and objects returned from indexing expressions:

matr1 = *matr2*[all()][rshift()]; // *matr1* and *matr2* are of type Matrix<>

In the first case, a new Matrix is created that contains the sum of the two argument matrices. This implies that after execution of this line, Matrix *matr1* contains data that are completely independent of those of *matr2* and *matr3*. In the second case, no new data are created. Instead, *matr1* defines a new *view* to the data of *matr2*. This means that *matr1* and *matr2* act as handles to the same data, but have a different interpretation of the Matrix indices.

It is useful to create a view when you refer to a given subset of a Matrix several times in your program. The following example demonstrates the definition of a view on a rectangular subset of a Matrix:

Program snippet:

```
iMatrix field(4,4);
iMatrix window = field[range(0,1)][range(2,3)];
field = 1;
window = 0;
cout ≪ "The window is" ≪ endl;
window.print();
cout ≪ "The total field is" ≪ endl;
field.print();
```

Output:

```
The window is
 0  0
 0  0
The total field is
 0  0  1  1
 0  0  1  1
 1  1  1  1
 1  1  1  1
```

In the case of simple subdomains, Matrix views are an efficient alternative to the use of domain specifications through the `where` command. Note however that such views are also a possible source of bugs, as Matrix indexing is often used without the aim of creating a view. In those cases, it is important to ensure that two independent dataspaces are used, using the assignment function `.assign()`

newMatrix.assign(*oldMatrix*[someIndex])

or alternatively the `.clone()` method

newMatrix = *oldMatrix*[someIndex].clone();

Section 9 explains in detail how to chose between these possibilities.

7 Domain specifications

7.1 The `where` command

The `where` command is a multi-dimensional extension of the classical `if` condition. Its aim is to constrain the execution of Matrix expressions to a certain subdomain. In the following example, a constant value is assigned to Matrix b at all element at which Matrix a has a value smaller or equal to 0.5:

Program snippet:

```
fMatrix a(3,3); iMatrix b(3,3);
forAll(a, IniRandom());
cout ≪ "a=" ≪ endl;
a.print();
where(a <= 0.5);
    b = 0;
elsewhere();
    b = 1;
endwhere();
cout ≪ endl ≪ "b=" ≪ endl;
b.print();
```

Output:

```
a =
 0.481   0.817   0.273
 0.520   0.267   0.017
 0.951   0.601   0.338

b =
 0  1  0
 1  0  0
 1  1  0
```

Although this problem could have been resolved by a simpler expression:

```
b = (a<=0.5).convert<int>();
```

it should make clear how the `where` command is applied. The argument to this command is a bMatrix or a so-called BoolMask (section 7.2). The expressions contained in a `where` block are subject to the domain restriction only when they concern matrices of the same size as the Matrix in the parameter of the `where` command. In all other cases, the domain restriction is ignored.

`Where` conditions are very often used in conjunction of mesh grids, in order to express a static domain restriction. The following command for example restricts the computation domain to the left half of the matrices:

```
fMatrix a(l,h);
where(a.meshGrid(1) < l/2);
...
```

7.2 Bool masks

The construction of a mesh grid and the computation of a Matrix that defines a domain restriction can be computationally intensive. For example, let us suppose that a programmer wants to restrict the computation domain to a circle:

```
...
where(sqr(a.meshGrid(1)-cx)+sqr(a.meshGrid(0)-cy) < r*r);
    a = ···
endwhere();
```

This operation involves the construction of 2 mesh grids, and the application of 6 algebraic/logical operations over the whole extent of the matrices. When the same `where` condition is used several times (for example inside a time-dependent loop), it is therefore useful to store the conditional expression:

```
...
    // store the conditional expression
bMatrix circle = sqr(a.meshGrid(1)-cx)+sqr(a.meshGrid(0)-cy) < r*r;
for (t=0; t<maxT; ++t) {
    where(circle);     //use of the stored condition Matrix
        a = ···
    endwhere();
}
```

The condition Matrix is internally stored by Vladymir in the form of a so-called BoolMask. This is a compressed format that optimizes both memory usage at evaluation speed of Vladymir expressions that are subject to the `where` condition. One of the great virtues of this format is that negation of the condition (as applied implicitly by the elsewhere() function) is a constant-time operation.

To short-circuit the bMatrix → BoolMask conversion, the condition Matrix can be directly stored as a BoolMask by the user. Thus, the declaration of Matrix `circle` in the previous example is advantageously replaced by the following declaration:

```
    // conversion bMatrix --> BoolMask
BoolMask circle = sqr(a.meshGrid(1)-cx)+sqr(a.meshGrid(0)-cy) < r*r;
```

7.3 Local `where` conditions

This section of the User Guide is yet to be written...

8 Input / Output

8.1 Vladymir I/O streams

The standard C++ I/O streams (cin, cout, cerr, clog) have been redefined in Vladymir in order to have a behaviour corresponding to the expectations Vladymir users most likely have. To understand why this is necessary, consider how the following would behave in a parallel execution, if cin and cout had not been redefined:

```
int number;
cout ≪ "Enter a number:  ";
cin ≫ number;
```

The ouput command on the first line, being executed on all processors, would result in the output of the string "Enter a number:", not only one time, but as many times as there are processors. Even worse, the input command on the second line will result in all processors but the first one being blocked. Indeed, given that they are not coupled with the input device, they will wait for user input eternally.

But as already said, the standard streams have been redefined to face this problematic. With the new definition, the above code executes correctly: the output message is displayed only once, and the input is broadcasted to all processors. As a conclusion, Vladymir users need not care about this question. It suffices to know that there was some kind of problem, but it has been solved.

A little attention must be spent however when defining file streams. The use of `fstream` and associated objects yields erroneous code at parallel execution, for the same reasons as the ones explained in the previous text. To face this problem, the new classes vfstream, vifstream and vofstream have been defined. Always use them in stead of the old fstream, ifstream and ofstream — in both programs intended to be run in serial and parallel: this doesn't add any complexity to your code (the new stream classes implement the same interface as the old ones) and keeps it parallelizable, just in case.

8.2 Serialization and unserialization of a Matrix

A Vladymir Matrix can be serialized and unserialized on any external device using the overloaded stream operators ≫ and ≪. Contrarily to the formatted output obtained by the method .print(), serialization produces a linear stream of data, containing in this order:

- Number of dimensions.
- Extent of the Matrix in each dimensions.
- The matrix data.

All numbers contained in the stream are separated from each other through a newline ("\n") character:

```
iMatrix randMat;
forAll(randMat, IniRandom());        // Random number initialization; see section
cout ≪ setprecision(8) ≪ randMat;    // Serialization on standard out
cout ≪ endl;
vofstream ofile("randMat.dat");      // IMPORTANT: always use vofstream
                                     //  instead of ofstream!
ofile ≪ randMat;                     // Serialization on output file
...
vifstream ifile("randMat.dat");      // Always use vistream instead of ifstream!
iMatrix newMatrix;
ifile ≫ newMatrix;                   // Unserialization from input file
```

While serialization/unserialization is very convenient and portable across machine architectures, it is quite unefficient when handling big matrices. In such a case, you might want to read about binary files in the next section.

8.3 Binary file I/O

The following code saves a Matrix in a binary file:

```
fMatrix a(3,3);
forAll(a, IniRandom());      // random initialization (section 4.3)
a.save("some-matrix.dat");   // saving a into a binary file
```

And this one loads the Matrix from the file:

```
fMatrix b(3,3);
a.load("some-matrix.dat");   // loading matrix from binary file
```

The data are written into the file in the same order as the serialized data:

1. The number of dimensions, as integer value (in the above example, it has the value 2).
2. The extent of the matrix in each dimension, as integer values (in the example, twice the value 3).
3. The data (in the example, 9 double precision real values).

Note that in the current version of Vladymir, the methods .load() and .save() are non-portable. First, they are implemented specifically for Unix(tm)-like systems. Second, on a parallel run they suppose that the underlying file system is distributed (NFS or alike).

8.4 Defining a new streamable device

This section of the User Guide is yet to be written...

8.5 Using Matlab(tm)

Vladymir is interactively connected to Matlab(tm) – that is, if you didn't switch off this possibility at compilation of Vladymir. This interoperability implies that data can be sent from Vladymir to Matlab(tm) and back. Furthermore, a Vladymir program can execute Matlab(tm) commands by submitting a command string. Any string is valid that would constitute a valid command at the Matlab(tm) prompt.

In the following example, a Matrix is inverted with the aid of Matlab(tm):

```
fMatrix anyMatrix;
...                                          // data initialization
toMatlab(anyMatrix, "anyMatrix");            // transfer data to Matlab(tm)
matlab().command("invMatrix = 1/anyMatrix"); // execute Matlab(tm) command
fMatrix anyOtherMatrix;
fromMatlab(anyOtherMatrix, "invMatrix");     // transfer data back from Matlab(tm)
```

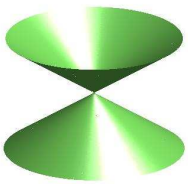
The command `fromMatlab()` transfers data back from Matlab(tm) to a Vladymir Matrix. There is no need for this Matrix to allocate memory previously, as memory allocation is automatic. However, the data types between the Matlab(tm) Matrix and the Vladymir Matrix must coincide. Note also that allocation of extra memory can be avoided when the Matrix delivered to `fromMatlab()` has been previously allocated the right shape.

Matlab(tm) will probably be used mostly as an engine for graphical output. Indeed, the graphics facilities are reasonably fast and extremely powerful for scientific purposes. As a small, admittedly not very useful illustration (I am short of ideas), let us look at the plot of a cone:

Program snippet:

```
iMatrix a(100,100,100);
a =  sqr(a.meshGrid(1)-50)+sqr(a.meshGrid(0)-50)-
     sqr(a.meshGrid(2)-50);
toMatlab(a, "cone");
matlab().command("isosurface(cone, 0)");
...
```

Output:



9 Copy semantics

This section of the User Guide is yet to be written...

10 Parallelism

10.1 Concepts

Vladymir programs are automatically parallelized (given, of course, that parallelization whas enabled at the compilation of Vladymir). For parallelization to work, it suffices to include the right header file (`pvladymir.h`) and, well, to run the program on multiple nodes (this is typically achieved through execution of the `mpirun` or `mprun` script). In particular, there is no need for precompilation or any other special treatment of the code. Parallelism is achieved dynamically at run time and is, at least when restricted to default parallelization strategies, completely transparent to the user.

The key to this simplicity (parallelism in its general formulation is an extremely delicate task) is that Vladymir encourages the programmer to a programming style that is inherently parallel. That is, computationally intensive operations are, hopefully, expressed through Matrix expressions acting on each element of the matrices. Vladymir implements a form of data parallelism: Matrix data are distributed over the nodes of a parallel machine, and the Matrix expressions are evaluated simultaneously on any of these nodes.

The underlying programming model is SPMD (single program multiple data). This means that a program that uses Vladymir is executed, without any changes, on all nodes of a parallel machine. Communication between the nodes is achieved through the message passing interface (MPI) standard.

10.2 Data managers

The data layout of the matrices over a parallel machine can be defined through the choice of so-called data managers. Data managers are policy classes that handle memory allocation, memory distribution on parallel machines, data parsing strategies and some I/O facilities.

The type of a data manager is chosen as a template parameter of the Matrix class. A data manager is constructed (or copy-referenced) at every construction of a Matrix. It is possible to explicitly construct a Matrix from a data manager (section 3.1).

A complete definition of the Matrix class template helps in understanding this issue:

```
template <typename T, template<typename U> class DMan>
    class Matrix;
```

The first template parameter is the type of the Matrix elements, and the second parameter the type of the data manager. The types `iMatrix`, `fMatrix` and `bMatrix` are really typedefs:

```
typedef Matrix<int, SerialDataManager>      iMatrix;
typedef Matrix<double, SerialDataManager>   fMatrix;
typedef Matrix<bool, SerialDataManager>     bMatrix;
```

You can of course write your own typedefs in order to instantiate other types of Matrices (as shown in section 3). The current implementation of Vladymir ships four types of data managers:

`SerialDataManager`	For serial execution of Matrix expressions. In a parallel run, the data are duplicated on every processor. This can be a reasonable choice when the volume of communication between processors grows too large for efficient parallelization.
`EquiBalancedDataManager`	For parallel execution of Matrix expressions. The data are evenly distributed over the processors of a parallel machine, slide by slide.
`ModuloDataManager`	For parallel execution. The data are distributed over the processors in small chunks, periodically. This enables a simple form of static load balancing (see section 10.4).
`DynamicDataManager`	The data manager used for doing dynamic load balancing. Note that instead of instantiating explicitly a Matrix on this data manager, you are better off using the `Group` facilities presented in section 10.5.

The `DefaultDataManager` is a typedef defined in `vladymir.h`:

typedef SerialDataManager DefaultDataManager

and in `pvladymir.h`:

typedef EquiBalancedDataManager DefaultDataManager

In the most general case, Vladymir users will write their own data manager to fit specific needs. It is out of the scope of this user guide to explain how this is done. However, reading the code of the default data managers might give a sufficient guideline for this task. If not, you are welcome to contact the author (section 1.1.3 on this topic.

10.3 Caveats

Automatic parallelization is an extremely powerful feature of Vladymir code, but it comes with some danger: writing unefficient, and, in some cases, even erroneous code. This section gives an overview on basic guidelines that ought to be respected when writing code for parallel execution with Vladymir.

10.3.1 Erroneous code

We start this discussion with the worse case, where an erroneous code produces undefined result or a result depending on the number of nodes on which it is executed. If we assume that there is no bug left in Vladymir itself, then the only situation in which such problems can appear is during access to external devices. Indeed, imagine a program naïvely accessing, say, a printer. During parallel execution, all code is duplicated on all nodes, so all nodes will try to do their print job. If they actually *have* access to the printer, this story results in a sort of concurrent print job which the printer is very probable to dislike.

The solution to this problem is to use Vladymir routines for access to external devices instead of traditional C++ functions (and I am saying nothing here about C routines of the `open()` and `write()` family which are *pure evil!*). Although the corresponding procedure is detailed in section 8, a small outline shall be repeated at this place. Three possible ways there are for correct access to external devices in Vladymir code:

1. Using Matrix methods (`load`, `save`, `rawData`, ...) for mapping of Matrix data from and to a storage device.
2. Use the standard stream objects(`cin`, `cout`, `cerr` and `clog`) or the Vladymir file streams (`vifstream`, `vofstream`) for inserting data into the standard streams.
3. Redefine new Vladymir streams for accessing your favourite exotic external devices.

10.3.2 Unefficient code

It is though probably much more frequent that a code based on Vladymir is simply unefficient. The simplest case of such a behaviour is when an important part of the computations is effected without the use of Matrix expressions. Plain C/C++ code without Vladymir matrices is simply duplicated on all processors and not parallelized. The result is a program running at the speed of a single processor.

A second possible reason is that the code, although written in a array-based style using Vladymir matrices, is slowed down by the need for communication between processors. Dependent of the underlying architecture, communication can be orders of magnitude slower than computations and must thus be kept very low. For illustration, consider a code in which data are often shifted along a given dimension:

```
fMatrix matr1(l,h);
matr1.assign(matr2[rshift(n)][all()]);
```

The author of this code snippet followed the guideline of section 12.3, choosing to apply the shift in the highest(left-most) dimension to avoid clustering of data. From a point of view of parallelism however, this choice is not optimal as it results in communication between processors (we assume that the data manager in this example is the default `EquiBalancedDataManager`, where the data are linearly and evenly distributed over the nodes). Thus, in the optic of parallel execution, it might be more efficient to apply the shift in dimension 0:

```
fMatrix matr1(l,h);
matr1.assign(matr2[all()][rshift(n)]);
```

In such a way that the data are mostly copied around in the memory space of single processors instead of being shipped through the interconnexion network.

As a last reason for unefficient program executions we can name the situation where the code is slowed down, not by interprocessor communication, but by the increased algorithmic complexity of the evaluation of expressions. This is a general bottleneck occurring when the number of processors is extensively large. Indeed, Vladymir is responsible for coordinating the communication between processors. Doing so is a non-trivial task that can consume a sensible amount of time when many processors are used.

10.4 Load balancing

This topic addresses the attribution of Matrix data to the nodes of a parallel machine, in order to obtain a balanced distribution of the computational load. The strategies for load balancing are determined through the choice of a data manager (see section 10.2). The need for appropriate load balancing appears for example at the execution of simulations on domain with non-uniform boundaries. In Vladymir, the computational domain must be bounded by a rectangular Matrix, which results in a bunch of idle data and a potential un-balance of the processor load (and a potential waste of memory; to learn how to avoid that, turn to section 12.1).

The default parallel data manager `EquiBalancedDataManager` performs no load-balancing at all. It the most common situations, even when the load is just slightly unbalanced, this is the most efficient choice.

Going a step further, one can choose a *static* load balancing strategies. This is a good choice when the distribution of the load is known in advance (*e.g.* in case of simple static boundaries), or when a heuristic method shall be applied to smooth out bad balance. The only such method that is implemented by default in Vladymir is contained in the `ModuloDataManager`, which distributes the data column-wise over the processors, column by column and in a cyclic manner. This strategy shows espacially good results when there is very little need for communication between nodes, or when the structure of communication is nonlocal.

When nothing else works, it is time to switch to *dynamic* load balancing, a topic covered in the next section.

10.5 Dynamic load balancing

10.5.1 Data organisation and load balancing

The data describing one cell in, for example, a cellular automata or lattice Boltzmann simulation are usually represented by multiple matrices. The matrices all contain values in such a way that the values with the same index in the matrices refer to the same cell. In database jargon, such a representation is called 'transposed' data.

As a consequence, all matrices that are part of the data set corresponding to the cells in the simulation have the same dimensions. In 'High Performance FORTRAN', such a relation between data points in arrays can be made explicit by declaring the arrays to be 'aligned'.

Load balancing in Vladymir is closely related to the group of matrices that form the data set of the application which describes all cells. A 'Group' class exists which knows the dimensions of the simulated environment. The key features of the 'Group' class are the ability to construct new matrices and to balance the load by redistributing the matrices across all PEs. A PE is a 'Processing Element', typically a CPU with memory and communication facilities. The 'Group' class has an interface to set the 'where' conditions independent from the global 'where' conditions. The 'where' conditions of the 'Group' class only apply to the matrices in the group in which they are set.

10.5.2 The 'Group' class

The API of the 'Group' class is shown below. The API has been defined in addition to Vladymir's existing API to provide methods that ensure that all requirements for load balancing are met.

The constructor, **Group(vector<size_t> extent)**, needs the dimensions of all matrices in the 'Group' as an argument.

New matrices containing values of any type can be constructed with **createMatrix()**. All created matrices have the dimensions which were given as an argument to the constructor. The newly created matrix will be part of the group.

Group Group(vector<size_t> extent)
template<class T> Matrix<T, DynamicDataManager> createMatrix()
void setMemoryMask(Matrix<bool, DynamicDataManager> &memMask)
void clearMemoryMask()
void where(BoolMask const &boolMask)
void elsewhere()
void endwhere()
void resetTime()
void balance()
void convergenceOnly()
void balanceOnlyOnceIn(int steps)

If some parts of a matrix will actually not be used, it is not needed to allocate memory for that part. The programmer can indicate which parts will be used, and which parts will not, by providing a Boolean matrix using the method **setMemoryMask(...)**. This mask can be removed with a call to **clearMemoryMask()**.

The methods **where(BoolMask const &boolMask)**, **elsewhere()** and **endwhere()** respectively set, invert, and remove 'where' conditions to all matrices in the group. These 'where' conditions need to be stored for the load balancing algorithms, which is why these methods exist as a 'wrapper' around the usual equivalents in Vladymir.

A call to **balance()** will let the group do a part of the work necessary to balance the load among the PEs. All matrices in the group will be redistributed accordingly. The computation of a new distribution is not achieved in one call, the call should be made once in every timestep.

There are several methods which allow the programmer fine-grained control over the load balancing process. The 'Group' class takes time measurements, starting at the moment of construction. With a call to **resetTime()**, the group will start measuring time from zero. If the programmer wishes to reduce overhead generated by load balancing, a call to **balanceOnlyOnceIn(int steps)** will force the group to do nothing when a call to **balance** is made for the specified number of steps between different attempts to balance the load.

10.5.3 Selecting Algorithms

There are three algorithms for load balancing to choose from: a global, a local and a regional algorithm. The programmer must choose which algorithm he or she wishes to use by using one of three subclasses of class Group: GlobalGroup, LocalGroup or RegionGroup. The use of the GlobalGroup is recommended.

10.5.4 Note on memory usage

The memory optimisation allows one to prevent allocation of memory for parts of a matrix on which no computations take place, also known as 'dead' space.

Without this, we have the problem that a node with a huge amount of dead space may start swapping, which proved to be deadly for performance in benchmarks. Considering that even a lot of dead space hardly generates any load, it is very likely that a lot of dead space will be accumulated in one PE. Thus, it is recommended that a lot of memory remains free in each PE to buffer dead space.

With the removal of dead space, the situation is actually reversed: if one host has a lot of dead space, it has lots of free memory. The heaviest loaded nodes are likely to have a lot of memory in use. Load balancing will spread out the domain and thus flatten out memory usage. Thus, we can drop the recommendation if there is lot of dead space. Load balancing will actually improve the memory situation in those cases.

It there is no dead space but a lot of cells that have almost no weight, the unoptimised situation will occur as memory for those cells must be allocated.

10.5.5 An example application

The following piece of code is a wave propagation simulation and can be found in the directory 'samples' distributed with the source. It demonstrates how to make use of load balancing in Vladymir.

```
#include <cmath>
#include <vector>
#include <vladymir/base/vladymir.h>

static const int SPACE = 0;
static const int WALL = 1;
static const int DEAD = 2;

typedef Matrix<double, DynamicDataManager> fMatrix;
typedef Matrix<int, DynamicDataManager>    iMatrix;

int main(int argc, char **argv)
{
    const std::size_t   h = 1000;
    const std::size_t   l = 1000;
    const int   maxT   = 5000;

    iniVladymir(&argc, &argv);

    std::vector<std::size_t> dimensions;
    dimensions.push_back(h);
    dimensions.push_back(l);

    GlobalGroup grid(dimensions);
    fMatrix     f[4], f_[4], psi;
    iMatrix     obstacle;

    // Create and initialise the obstacle matrix
    obstacle = grid.createMatrix<int>();
    obstacle[range(0, h / 2)]         = DEAD;
    obstacle[h / 2 + 1]               = WALL;
    obstacle[range(h / 2 + 2, h - 1)] = SPACE;

    // Tell the group not to allocate memory for
    // unused datapoints.
    grid.setMemoryMask(obstacle != DEAD);

    // Create and initialise all other matrices.
    for (int i = 0; i < 4; ++i)
    {
        f[i]    = grid.createMatrix<double>();
        f_[i]   = grid.createMatrix<double>();
        f[i]    = 0.0;
    }
    psi = grid.createMatrix<double>();

    // Tell the group to stop load balancing once
    // the system has reached a convergent state.
    grid.convergenceOnly();

    // Let time start at this moment.
    grid.resetTime();
    for(int i = 0; i < maxT; ++i)
```

```
    {
        /*
         * Collision ...
         */
        psi.assign(f[0] + f[1] + f[2] + f[3]);

        grid.where(obstacle == SPACE);
            // Normal dynamics
            f_[0].assign(0.5 * psi - f[2]);
            f_[1].assign(0.5 * psi - f[3]);
            f_[2].assign(0.5 * psi - f[0]);
            f_[3].assign(0.5 * psi - f[1]);
        grid.endwhere();
        grid.where(obstacle == WALL);
            // Boundary condition: bounceback
            f_[0].assign(- f[2]);
            f_[1].assign(- f[3]);
            f_[2].assign(- f[0]);
            f_[3].assign(- f[1]);
        grid.endwhere();

        // Boundary condition: generation of wave.
        f_[0][0] = 0.0;
        f_[2][0] = sin(0.1 * (double)i);

        /*
         * Propagation ...
         */
        f[0].assign(f_[0][rshift()]);
        f[1].assign(f_[1][all()][rshift()]);
        f[2].assign(f_[2][lshift()]);
        f[3].assign(f_[3][all()][lshift()]);

        // Load balancing
        grid.balance();
    }
    return 0;
}
```

11 Extending Vladymir

11.1 Defining new operators

This section of the User Guide is yet to be written...

11.2 Accessing raw data

This section of the User Guide is yet to be written...

12 Optimizations

12.1 Dead data

In Vladymir, the fundamental data type is a rectangular multi-dimensional Matrix. This can be a partiacularly annoying fact when the Matrix shall represent a spatially extended physical system which has *not* nice rectangular boundaries. This invariably leads to a set of unused data between the domain

boundary and the Matrix box. Typically, the ratio of unused data to the total number of data increases with the number of dimensions of the Matrix. In 3D, matrices with a number of active cells as low as 20% are no rarity.

From the point of view of execution speed, this unefficient use of memory space is not a problem, as the computational domain can be efficiently restrained by the use of `where` conditions (section 7). The real problem concerns the use of memory: for memory consuming application, it is interesting to avoid memory allocation for those unused data areas.

A simple way of doing so is to define *dead data*, *i.e.* cells that take formally part in the Matrix, but for which no memory is allocated. During evaluation of Matrix expressions, the presence of dead data (in any one of the involved matrices) has the same effect as the specification of a `where` condition. The corresponding data range is simply excluded from the computations. During output or data transfer to Matlab(tm), dead data are converted to real data containing the value `NaN`. Dead data regions are specified at the construction of a Matrix through a BoolMask as the right-most argument to the constructor. This is illustrated by the following example:

`Program snippet:`

```
iMatrix dummy(3,3);
iMatrix field1(3,3, dummy.meshGrid(1)<2);
field1 = 1;
field1.print(); cout ≪ endl;
iMatrix field2(3,3); field2 = 0;
field2.assign(field1);
field2.print();
```

`Output:`

```
0  0  NaN
0  0  NaN
0  0  NaN

0  0  1
0  0  1
0  0  1
```

We remark that especially in presence of dead data, the data structure used for data allocation doesn't ressemble a regular array at all. It is rather a list of memory chunks that cover exactly the bulk of irregular domains. Nevertheless, this is transparent to the Vladymir user who gets a Matrix abstraction of the data. This abstraction can be a very powerful tool for software engineering, as it replaces iterations over list elements by simple Matrix expressions.

An interesting application domain is the definition of local grid refinements. One can represent the computational field by successive layers of which every one has a different degree of refinement. The field of the simulation is a superposition of those layers. Although all layers formally span the complete domain, thanks to the use of dead data they really allocate memory only for those parts of the domain that implement the appropriate degree of refinement.

12.2 Load sections

This feature is yet to be implemented in Vladymir...

12.3 Data alignment

This section of the User Guide is yet to be written...

12.4 Reference or copy

This section of the User Guide is yet to be written...

A Arithmetic and logical functions

A.1 Overloaded operators

A.1.1 Arithmetic binary functions

operator	associated functional
operator+	plus
operator−	minus
operator∗	multiplies
operator/	divides
operator%	modulus

Example:

```
fMatrix a, b, c;
a = b + c;
a = b + 3.14159;
a += b;
```

A.1.2 Logical binary functions

operator	associated functional
operator==	equal_to
operator! =	not_equal_to
operator<	less
operator>	greater
operator<=	less_equal
operator>=	greater_equal
operator&&	logical_and
operator||	logical_or

Example:

```
bMatrix a, b;
iMatrix c;
a = a || b;
a = c < 1;
```

A.1.3 Bit-level binary functions

operator	associated functional
operator&	bitwise_and
operator|	bitwise_or
operatorˆ	bitwise_xor

Example:

```
iMatrix a, b c;
a = b | c;
c &= 2;
```

A.1.4 Unary functions

operator	associated functional
operator+	plus
operator−	minus
operator!	not
operator∼	bitwise not

Example:

```
fMatrix a, b;
a = -b;
```

A.2 Math functions, mostly from cmath

A.2.1 Unary functions

function	signification
acos	arc cosine
asin	arc sine
atan	arc tangens
ceil	round up
cos	cosine
cosh	hyperbolic cosine
exp	exponential function
fabs	absolute value
floor	round down
log	log of base e
log10	log of base 10
sin	sine
sinh	hyperbolic sine
sqr	square (not in cmath)
sqrt	square root
tan	tangens
tanh	hyperbolic tangens

Example:

```
fMatrix a, b;
a = sqrt(b);
```

A.2.2 Binary functions

In the following cases, first argument is a Matrix, the second argument a scalar.

function	signification
pow	power
fmod	(real valued) modulus

Example:

```
fMatrix a, b;
a = pow(b, 2.3);
```

In the following, at least one argument must be a Matrix, and the other a Matrix or a scalar.

function	signification
minVal	Minimum value (not in cmath)
maxVal	Maximum value (not in cmath)

Example:

```
fMatrix a, b, c;
a = minVal(b, c);
```

A.3 Reduction functions

Reduction functions reduce a (sub-) matrix to a scalar value.

function	signification
reduceMax()	maximum value
reduceMin()	minimum value
reduceSum()	sum
reduceProd()	product
reduceLAnd()	logical and
reduceLOr()	logical or
reduceLXor()	logical xor

Example:

```
fMatrix a(3,3);
double scalar;
scalar = a.reduceSum();
```

B Frequently asked questions